Middleware for Wireless Sensor Networks: An Outlook

Luca Mottola · Gian Pietro Picco

Abstract In modern distributed computing, applications are rarely built directly atop operating system facilities, e.g., sockets. Higher-level *middleware* abstractions and systems are often employed to simplify the programmer's chore or to achieve interoperability. In contrast, real-world wireless sensor network (WSN) applications are almost always developed by relying directly on the operating system.

Why is this the case? Does it make sense to include a middleware layer in the design of WSNs? And, if so, is it the same kind of software system as in traditional distributed computing? What are the fundamental concepts, reasonable assumptions, and key criteria guiding its design? What are the main open research challenges, and the potential pitfalls? Most importantly, is it worth pursuing research in this field?

This paper provides a (biased) answer to these and other research questions, preceded by a brief account on the state of the art in the field.

1 Introduction and Motivation

Wireless sensor networks (WSNs) are finding their way into the mainstream. Networked embedded systems sensing from, and acting on, the environment are increasingly employed as a bridge between the physical and digital world. To achieve

Luca Mottola
Swedish Institute of Computer Science
E-mail: luca@sics.se
WWW: www.sics.se/˜luca

Gian Pietro Picco
Department of Information Engineering and Computer Science, University of Trento, Italy
E-mail: gianpietro.picco@unitn.it
WWW: disi.unitn.it/˜picco

this, significant financial resources and man power have gone in devising efficient technical solutions at all levels, from the hardware up to the application layers. Among these, middleware abstractions and systems have been in the focus of several research efforts [9, 13, 23, 26, 28].

However, unlike mainstream distributed computing, middleware is not always referred to as such in WSN research. Current designs favor architectures where the WSN stack is highly application- or even deployment-specific, rather than application-agnostic as usual. This is due to the target hardware: traditional architectures are simply not affordable given the scarcity of computing and communication resources. For better resource utilization, layers blur and blend together, to the point that placing a middleware *layer* in a WSN design becomes difficult, even conceptually.

As a consequence, a number of solutions are available in WSN research that would be traditionally termed as middleware technology, as Section 2 illustrates, but often are not explicitly named so. These solutions cover a variety of aspects: from programming abstractions to component models and distributed protocols. Yet, most of the real-world deployments reported in the literature do *not* leverage this functionality, and resort to implementing highly customized mechanisms right atop the operating system (OS).

To understand why this is the case, and what are the research challenges involved, the rest of the paper unfolds as follows. In Section 3, we briefly report on our direct experience in applying WSN middleware in a real deployment, articulating on what it takes to move WSN middleware out of the research lab into the physical world and on the benefits that can be reaped by doing so. Section 4 distills what we believe are the most significant open research challenges in this field; our target readers include both those who *build* WSN middleware and those who instead *rely* on it. In Section 5 we widen the scope of the discussion to considerations concerned with the research community and industry

```
SELECT AVG(light), AVG(temp), location
FROM sensors
SAMPLE PERIOD 2 s FOR 30 s
```

Fig. 1 Monitoring bird nests using TinyDB.

at large, with the intent to share insights useful towards a long-term research strategy. We conclude in Section 6 with a note about two crucial, overarching needs concerning WSN middleware, namely, how to *i)* demonstrate its applicability in the real world, and *ii)* raise awareness about the peculiarity of its design and implementation w.r.t. mainstream systems.

2 A Concise Look at the State of the Art

The notion of middleware manifests itself with various facets in WSN research [9, 13, 26, 28]. For instance, a number of programming abstractions [23] provide functionality commonly deemed as "middleware", although this term does not appear in papers and documentation. Moreover, several WSN component-based approaches exist, in a sense akin to mainstream component-based middleware. Finally, even the functionality shipped with the OS is sometimes referred to as "middleware", mainly as a result of the different role taken by WSN OSes compared to traditional networked systems. We describe next three examples corresponding to these flavors of WSN middleware.

2.1 WSN Programming Abstractions: TinyDB

Many existing WSN programming abstractions can be regarded as providing middleware functionality [27] at various levels of abstraction. A first distinction might be drawn as to whether a given system enables only the specification of actions taken by individual devices, or instead allows one to program the network as a whole—the latter often referred to as "macroprogramming". Nevertheless, we observe that this classification is too coarse-grained and fails to capture other fundamental characteristics of existing approaches (e.g., the communication or computation scope a given abstraction enables) or its intended use (e.g., as standalone programming system vs. building-block abstraction). In previous work, based on the state of the art, we identified several dimensions worth considering in this respect [23].

As an example, we describe here TinyDB [20], a query processing system whose focus is to optimize energy consumption by controlling where, when, and how often data is sampled. SQL-like queries are submitted by the user at a base station where the application intelligence resides. There, queries are parsed, optimized, and injected into the WSN. Upon reception of a query, a WSN node processes the corre-
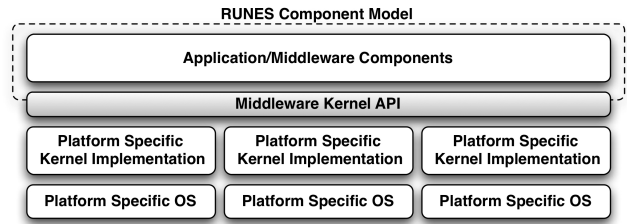

Fig. 2 RUNES middleware architecture.

sponding data requests by gathering readings as needed, and funnels the results back to the base station.

Figure 1 shows an example to monitor the presence of birds in nests [20]. The data model revolves around a single `sensors` table that logically contains one row per node per instant in time, and one column for each data type the node can produce (e.g., temperature or light). In the example, the average light and temperature around a nest are gathered every 2 s for a total of 30 s. The `SELECT`, `FROM`, and `WHERE` clauses have the usual SQL semantics, while `SAMPLE PERIOD` specifies the query rate and lifetime.

Data collection applications are easily expressed with TinyDB, as its declarative abstraction helps programmers focus on the data to retrieve without specifying how to do so. As a layer sitting between the OS (i.e., TinyOS in this case) and the application developer, TinyDB is sometimes referred to as WSN middleware [9]. Moreover, to some extent it provides interoperability between the WSN and backend systems, as it also includes libraries to store the results of TinyDB queries on standard DBMSes.

2.2 Component-based WSN Middleware: RUNES

Enabling dynamic re-configuration is one of the main motivations for component-based designs in WSNs [19, 21, 24]. The need arises mostly in application scenarios with changing requirements, where the system must adapt on the fly. The ability to individually deploy units of functionality thus becomes a critical need for flexibility and resource efficiency, as replacing the entire binary running on nodes is an energy-intensive operation and requires interrupting the normal system operation.

The RUNES middleware [11] is representative of this type of approach, and also provides interoperability among some hardware/software platforms. Figure 2 depicts the RUNES architecture. The foundation is a component-based programming model, available to programmers through a middleware kernel API. This interface is used to build a composition of middleware and application-level components offering the necessary functionality. Rather than providing a monolithic "layer" as in TinyDB, orthogonal middleware features are factored out into self-contained components, se-

lectively deployed according to resource constraints and application requirements.

For example, some devices might require only a basic communication component providing unreliable messaging, whereas others might require reliable communication, realized on top of the base components. However, these needs may also arise for the same device at different times, based on changing application requirements. To address this need, the RUNES middleware allows the set of components to be updated at run-time, with the platform-specific kernel implementation managing such dynamic reconfiguration.

2.3 Operating System Libraries: TinyOS and nesC

Operating systems for WSNs are typically simple, providing basic mechanisms to schedule concurrent tasks and access the hardware. This sharply contrasts with conventional OSes, which are rather complex and support functionality such as memory protection and user interaction. Moreover, the communication constructs built into WSN OSes are usually 1-hop only; more complex patterns must be realized as intermediate functionality between OS and application, effectively making a case for a "middleware" layer [18]. In this respect, a representative example is TinyOS [15] and the accompanying nesC language.

nesC is an event-driven programming language derived from C. Applications are built by interconnecting *components* that interact by providing or using interfaces. The functionality encapsulated in each component describes the actions a node is to perform. Interfaces list one or more functions, tagged as *commands* or *events*. Commands are used to start operations; events are used to collect the results asynchronously. TinyOS manages the scheduling of commands and events, as well as their interactions with the hardware.

The only communication functionality built into TinyOS is the Active Message API. This allows messages to be tagged with an identifier specifying which component must process them upon reception, in a way similar to TCP/UDP ports, although limited to 1-hop broadcast and unicast. In this context, Active Messages play a role similar to sockets in mainstream distributed computing, by providing a basic building block enabling the development of higher-level functionality. Multi-hop communication (e.g., data collection and dissemination) is realized atop Active Messages [2].

3 A First-Hand Example: The TeenyLIME Middleware

To provide the reader with a concrete feel of what can be attained by WSN middleware and the research challenges entailed, in this section we focus on a system called TeenyLIME [3, 12], for which we have first-hand experience both w.r.t. its development and real-world use [7, 8].
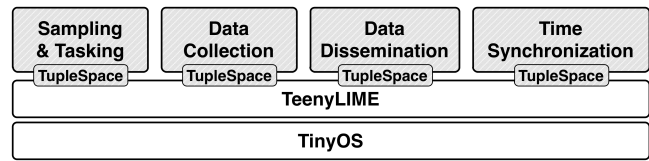

Fig. 3 The TeenyLIME-based architecture used in [8].

TeenyLIME is based on the tuple space abstraction, a memory space where processes read/write data in the form of tuples, shared among nodes within radio range. Besides operations to insert, read, and withdraw tuples, reactions allow for asynchronous notifications when data of interest appear in the shared tuple space. WSN-specific features are provided, e.g., to maintain system information about neighbors. TeenyLIME's abstractions essentially replace the 1-hop message passing provided by nesC with 1-hop data sharing, and are useful to develop both application- and system-level mechanisms (e.g., routing or time synchronization).

The effects of this change in communication paradigm are many. From a qualitative point of view, the sharp decoupling provided by data sharing boosts reuse within and across applications, and leads to code that is inherently less complex. From a quantitative point of view, the code written by the developer is *significantly* more concise. For instance, we found the size, in lines of source code, of a simplified HVAC (Humidity, Ventilation, Air Conditioning) application to be 25-70% smaller with TeenyLIME than with nesC [12].

Our initial findings with TeenyLIME were confirmed during its first real-world deployment, a structural health monitoring system in a medieval tower in Trento, Italy [8]. Figure 3 shows the architecture we used for such system, based on a TinyOS implementation of TeenyLIME targeting TMote Sky nodes [25]. Across the development of the different functionality, we observed code reductions between 50% and 80% w.r.t. alternatives in the TinyOS distribution.

Nevertheless, a reduction in the source code size does not imply only a reduction of the programmer's effort; it also yields smaller binary code, and therefore allows one to pack more functionality in the memory-tight WSN nodes. For instance, the alternatives provided by TinyOS for data collection, data dissemination, and time synchronization would not have fit together in the 48 KB available on TMote Sky nodes—despite the code memory taken by TeenyLIME. These considerations prompted us to use it in other deployments, e.g., closed-loop adaptive control of light levels in an operational road tunnel [7].

In our experience, WSN middleware is therefore not a luxury: it is a *need*, in that it not only simplifies development, but also enables an efficient use of program memory. Nevertheless, harvesting these benefits does not come for free. For instance, although we essentially kept the same

API, the initial implementation of TeenyLIME [12] underwent significant changes to meet the requirements of our structural health monitoring application [8]. Modifications were required, e.g., to provide efficient memory management for the high data rates involved, and to tailor the communication stack to the specific hardware platform—issues that would not have surfaced had we stopped at simulation and tabletop experiments.

4 Open Research Challenges

Designing and implementing middleware for WSNs is a research endeavor with many open challenges. In the following, we focus on those we deem particularly significant, based on our analysis of the state of the art and our own first-hand experience. Moreover, these challenges are only going to be exacerbated when WSNs will become key elements of larger pervasive infrastructures such as *Internet of Things* (IoT) or *Cyber-Physical Systems* (CPS). Therefore, we also hint at how these scenarios will impact WSN middleware.

Finding the right abstractions—for the right application. In today's WSNs, the functionality provided by middleware is biased towards sense-only, fixed-node applications:

- most programming abstractions, component-based systems, and OS-level libraries, are geared towards a many-to-one communication spanning the entire WSN. This reflects the needs of sense-only applications w.r.t. data collection, but it is ill-suited for sense-and-react scenarios (e.g., CPS ones), key to WSN integration into industrial practice. Here, communication is also one-to-many and many-to-many, and focused on subsets of nodes distinct by role, e.g., sensor vs. actuator. Abstractions expressly supporting these scenarios are needed;
- despite the many proposed applications where WSN nodes are attached to mobile entities [17], and the many networking solutions supporting them, very few middleware systems are designed with mobility in mind. A notable exception is Impala [19], which provides mechanisms for delay-tolerant, network-wide reprogramming. The general trend is however to assume the WSN as entirely composed of fixed nodes. This is surprising, given the vast literature on middleware for mobile computing [22]. IoT scenarios, and specifically their emphasis on the user and her interaction with the environment, are likely to bring mobility back at the center of the stage of middleware research.

Finding the right abstractions—for the right developer. As discussed in Section 2, WSN middleware exists at different levels of abstraction. Nevertheless, most of the programming activity in real-world deployments takes places at the lowest possible level—right atop the OS. We believe the answer to the question whether WSNs are going to become commonplace will be determined by the ability to empower the right user with the right programming abstraction. We identify three possible types of developer:

- *domain experts* are usually scientists or engineers, typically quite skilled with basic IT tools (e.g., databases and spreadsheets). Their main concern is to have "good data", e.g., without noise and accurately timestamped. They require high-level abstractions simplifying the configuration of the WSN at large, possibly allowing one to define its software architecture based on pre-canned functionality. They see the WSN as a macro-component delivering a useful service, and therefore require the middleware platform to hide distribution to the maximum extent. The aforementioned TinyDB system is an attempt in this direction. However, complexity often strikes back, as the conceptual gap from the operation of the individual devices to a network-wide perspective is significant. Hence, WSN middleware pursuing these goals tends to be very rigid and difficult to adapt to different needs;
- at the other extreme, *WSN geeks* are those who, today, develop the software for, and often deploy, the WSN. They are typically skilled at embedded systems programming and protocol design, and can deal with complex languages and systems. Their main concern is to have "good yield", i.e., to efficiently funnel as much data as possible to the collection point. Therefore, they need abstractions that, albeit low-level, simplify intra- and inter-node communication and computation, supporting both application- and system-level functionality. Moreover, they need APIs exposing knobs for tuning the system performance. These features are currently provided by the OS, whose programming model however easily distracts developers from application goals and into low-level details and protocols. The resulting implementations become entangled and difficult to maintain and to port, making the case for a lightweight middleware layer;
- *WSN technicians* are the middle ground between these two extremes—one that does not really exist today. We envision these developers with good IT technical background and expertise, able to build systems in addition to using them, but not necessarily acquainted with the intricacies of WSN innards. We believe that the vision of domain experts programming WSNs without help is realistic only in simple scenarios; more likely, a WSN technician will help deploy the system and customize it to application-specific requirements. To achieve this goal, this kind of developer will require more visibility into the system operation compared to domain experts. On the other hand, WSN technicians most likely will not develop system-level or hardware-specific functionality—the turf of WSN geeks—and will also need mechanisms for tuning performance based on high-level goals instead

of low-level knobs. Unfortunately, few approaches currently support this developer type.

Language vs. middleware, monoliths vs components. The quest for abstractions and expressive power is only half of the coin. The other half is their realization in a way that does not sacrifice performance. WSNs are applied in environments and scenarios with significantly different requirements (e.g., static vs. mobile, low vs. high data rate, low vs. high reliability). The need to meet the tight resource bounds brings the additional requirement of a flexible run-time: a one-size-fits-all solution with many unused features may not be practical, given that resources are limited.

We note instead that many existing systems are designed as *languages* [23]. In other words, they are meant to be a full replacement of the underlying OS-based language (e.g., nesC) with a higher-level one. This approach places great expressive power in the hands of the programmer, often at the expense of flexibility. The user is shielded from low-level details, and this prevents many opportunities for customizing the underlying run-time, which is often monolithic and cast in stone. If the proposed abstractions are not suited for the application at hand, the developer must resort to (often completely) different ones, or go back to implementing directly on top of the OS facilities.

The OS typically provides a great degree of componentization. As already noted in the case of TinyOS, the set of run-time components atop the OS effectively plays the role of a WSN middleware, providing a set of *network abstractions* [18] offering communication services for application development. We maintain that this view, which nonetheless enabled many of the stand-alone sense-only deployments in the literature, is not sufficient when tackling the more complex CPS and IoT scenarios. Here, the presence of in-network actuation and/or richer end-user requirements significantly complicates development. While WSN geeks may be well-suited to develop stand-alone WSN applications, these more sophisticated environments will require collaboration with domain experts and WSN technicians, who cannot deal with the lowest abstraction levels.

Striking a balance among these conflicting requirements is challenging, yet attempts are ongoing. For instance, in the make*Sense* [1] project, we target a system featuring *i)* an *extensible* macro-programming language, *ii)* a foundation of run-time components. This will allow developers to customize the programming model based on application requirements, by composing abstractions on a per-application or even per-deployment basis. Only the run-time components supporting the abstractions needed in a given scenario will then be automatically woven together by a dedicated compiler.

What about cross-layering? Since the inception of the field, WSNs were expected to challenge traditional layered architectures through cross-layer designs, for greater efficiency and better resource utilization [4]—a goal achieved only partially [10]. Cross-layering affects directly WSN middleware, and brings several technical challenges.

Firstly, as layers blend together, it becomes difficult to place a middleware "layer" anywhere in the stack. In traditional architectures, the positioning of middleware is quite well-defined: above the OS-provided network stack, below the application, and well-decoupled from both. As we already mentioned, in WSNs there is really no OS-provided network stack. Moreover, the application layer often must include custom communication functionality. Therefore, a commonly-agreed architecture including a WSN middleware is difficult to identify, and defining its interfaces towards the application often results in ad hoc solutions. Put in perspective, this situation will render standardizing WSN architectures, let apart middleware, quite a challenge.

Secondly, although access the low-level knobs is considered key to optimize resource consumption [4], this aspect is mostly neglected by the state of the art, partly because of the emphasis on monolithic approaches. However, striking a balance between flexibility and complexity in providing access to low-level features is probably one of the toughest, yet most important, problems in WSN middleware.

Enabling reliable, predictable implementations. Today's development process for WSNs essentially consists of trial and error through the design, implementation, and deployment phases. To some extent, this is unavoidable: given the interactions between the WSN and the environment, and often the unpredictability of the latter, it is impossible to anticipate all situations the system may have to deal with. Moreover, gaining an understanding of the root causes of failures is hard, due to the resource limitations of devices. Nevertheless, this ad hoc approach is very effort-demanding, ultimately hampering commercial adoption of WSNs.

Middleware may alleviate this issue by providing hooks to make the system behavior verifiable, both statically and at run-time. Monitoring tools integrated with the middleware may help developers diagnose problems and better understand their causes. We maintain that this functionality finds its natural place within the middleware rather than in the OS or at application level. The former would be too detailed to enable practical verification or debugging, while the latter option would severely limit re-use.

Moreover, upon failure, current middleware lets the WSN break down in unpredictable ways, as the run-time support provides no guarantees in these situations. Nodes running out of battery power, for instance, are eventually recognized and excluded from processing, but no bounds are provided w.r.t. when this happens. Transient faults (e.g., incorrect sensors readings) are usually not considered. Software errors are often fatal, yielding an erratic node behavior. To make things worse, faults at given nodes often affect others, caus-

ing a "domino" effect that ultimately renders the WSN unusable. These issues will become more and more important as WSNs become part of safety-critical systems. WSN middleware should provide *known* failure modes, along with tools and abstractions helping developers to understand the system behavior in these exceptional circumstances.

Supporting multiple concurrent applications. In existing real-world experiences, the WSN is always designed, implemented, and deployed with a single application in mind. On the contrary, traditional networks rely on a common, application-agnostic foundation of network protocols and services, serving the needs of heterogeneous applications.

This, however, is not necessarily going to remain the case. As WSNs become commonplace, it is natural to think of a WSN node running multiple applications. For instance, a node equipped with a temperature sensor may originally be programmed to report its readings to a fire application running in an office building. However, an HVAC application may be later deployed in the same building, relying on temperature readings to ensure the occupants' comfort. IoT scenarios exacerbate the problem, as they often regard WSN nodes as "clean-slate" components whose behavior can be transiently redefined according to (multiple) users' needs.

WSN middleware plays a critical role in enabling the scenarios above, which require functionality such as resource virtualization to control concurrent access to resources, on-the-fly reprogramming to enable on-demand deployment of multiple applications, and fair management of communication facilities shared across multiple tasks. These mechanisms are often delegated to the middleware layer, to keep the OS simple and enable re-use across applications. However, very little work exists in defining proper APIs and dedicated underlying mechanisms to provide such functionality.

Joining the flow: Integrating WSNs into the mainstream. We already mentioned that WSNs are often stand-alone sense-only systems. The data they gather is funneled to one or more collection points, and from there made available to the external world, typically in an ad-hoc fashion. In essence, WSNs are sharply decoupled and left at the periphery of the system: the "intelligence" resides outside the WSN and the system perceives the latter simply as a data source.

On the other hand, many popular scenarios rely on a different vision where the WSN actually hosts a significant part of the system intelligence. For instance, CPS are often thought as enforcing control laws in-network, with actuators cooperating to close the loop based on the information gathered by surrounding sensors. IoT scenarios often encompass complex peer-to-peer interactions among heterogeneous embedded devices. This may possibly rely also on communication with functionality external to the WSN.

Business processes are another example, which we are considering in the make*Sense* project. Industry has several solutions to describe, implement, and operate business processes: the Business Process Modeling Notation (BPMN) [6] and related technologies are examples. In this context, the option of interacting with WSNs in an ad-hoc fashion is not viable. WSN nodes and their capabilities must become first-class citizens in business process languages. The abstraction level must be such that a process modeler can specify the appropriate work-flow without being too distracted by low-level details concerned with the WSN. Unfortunately, these aspects have hitherto received little attention.

"How good is my middleware?" In the current practice, this question is often answered only from a performance standpoint (e.g., by evaluating the underlying protocols, or focusing on the memory and computational overhead) and typically only through simulation—both a relic of what commonly done by the networking and systems communities. However, this has at least two negative effects:

- the impact on programming practice is overlooked. Performance is important, but increasing productivity and the quality of the resulting implementations should also be a major goal—the defining one, actually. These aspects are currently largely ignored. Moreover, they are inherently difficult to assess, let apart quantitatively. At present, the only metric used is the number of lines of code, which is questionable as an indication of programming effort, and makes it impossible to compare approaches based on different programming paradigms. Finer-grained metrics, tailored to the specificity of WSNs, are sorely missing;
- simulations are only a very rough approximation: the gap w.r.t. a real-world deployment is significant. For instance, fluctuating link qualities have an impact on the system performance that current simulation models cannot reproduce. As a result, because of the lack of evidence that WSN middleware actually works in a real environment, very few of them are used in real-world deployments. This further hinders the field, by limiting the necessary feedback from domain experts and the in-field experience researchers would gain in the deployment process, as we further elaborate in Section 6.

5 Tactics vs. Strategy and Potential Pitfalls

Here we concern ourselves with considerations less tied to specific technical challenges, and more to "external" factors coming from the research community and industry at large. Our intent is to highlight some issues that should be considered when planning a long-term research "strategy"— opposed to the "tactics" possibly necessary to survive in the short-term—along with potential pitfalls.

Hardware and OS: Cozy or adventurous? WSN research has been ignited by the availability of cheap hardware platforms along with basic OS services, e.g., the MICA [14]

node and TinyOS. Subsequent developments in both radios and MCUs spurred major leaps for the field at large (i.e., not just middleware). Nevertheless, a frequently-heard comment is that we are currently "stuck" with TMote Sky-like nodes and TinyOS. Thus, it is fair to ask to what extent settling on this platform still enables significant progress. On the other hand, one could argue that, given the trends in miniaturization, more resources (e.g., memory, energy, and bandwidth), will be packed in the same hardware footprint. Is it then still meaningful to talk about resource limitations in WSNs?

On one hand, resource limitations are one of the defining features of WSNs: if one removes, say, memory limitations, a number of "challenges" in WSNs cease to exist. History has also shown that, along with more power within the same form factor, we will also see today's power in smaller devices—likely to enable new applications, pushing the envelope of what one can do with resource-constrained devices. Some of these challenges are, however, really system issues, where problems are essentially determined by hw/sw idiosyncrasies. On the other hand, one of the key ideas put forth by WSNs is the paradigm shift in enabling sensing (and/or actuation) in a distributed fashion: the issues of how to design and develop massively decentralized systems that behave as a coherent component of a bigger system is still open and does not go away with hardware enhancements.

Unlike with our everyday personal computers, one cannot just buy a more powerful model and use it with the same OS and applications. For instance, the most common MCU in WSNs today is the TI MSP430, specifically the variant with 48 KB of code memory and 10 KB of RAM aboard the TMote Sky. Other variants of MSP430 exist with more code memory, e.g., 96 KB and more. Leveraging this improvement (which is, all things considered, rather limited) may not be immediate, as we recently learned in an ongoing deployment where the application demanded more code memory. These MSP430 variants are not directly supported by TinyOS: using them requires changing portions of the TinyOS source, and in turn our own TeenyLIME middleware. Once more, developing WSN middleware entails challenges commonly not found in mainstream middleware.

"Have you ever heard of ZigBee?" ZigBee [29] is an industry-promoted standard for low-power wireless networks that specifies how applications can access the network stack (e.g., as discovery, addressing, security services) and configure it according to pre-defined, domain-specific profiles. In a sense, it provides a middleware simplifying application development. ZigBee is successfully used in many applications, especially in the automation landscape, building on the IEEE 802.15.4 standard for physical and MAC layers.

Academic WSN research and ZigBee appear to intersect only seldom, if at all. In part, this is a consequence of the platforms chosen. The CC2420, the radio chip most employed in WSN research because of cost and simplicity

of use, despite being 802.15.4-compliant, does not implement the MAC layer in hardware. This turned into a bonanza for researchers (free and almost compelled to experiment alternatives) that spurred a wealth of research on low-power MAC protocols, and also enabled deployments whose characteristics (e.g., bursty traffic or mobile nodes) do not match the assumptions of ZigBee. However, this progressively caused industry to lose interest in academic WSN research, as compliance with standards, especially when it comes to security, is key to industry applications.

As a consequence, the WSN notion appears today to be interpreted in two very different ways. Roughly speaking, in industry it often implies the use of ZigBee-compliant devices, whose network and middleware layers are rigidly defined, and where innovation occurs only in developing new application-specific hardware. In academia, it often implicitly assumes the use of a TMote-like device and TinyOS, without reference to a specific network/software stack.

It is clear what is the best playground for a middleware researcher. Nevertheless, it is also clear where the commercial value is: in our experience with big companies, their interest fades quickly after the issue of ZigBee compliance w.r.t. research systems is unveiled. Choosing the field to play determines directly the chances of real-world impact.

Is interoperability an issue? Interoperability is one of the defining features of mainstream middleware, but so far we barely touched upon it. To the best of our knowledge, there is really no WSN middleware available for more than one OS. The reason is that in the current state of the art interoperability is essentially an artificial problem. There is simply not enough of an installed base to justify an investment in a layer that papers over OSes, enabling nodes supporting different systems to coexist.

Things are, however, slowly changing. On one hand, virtual machines for WSN nodes are making a comeback after the early days. For instance, Darjeeling [5] provides a Java VM for commonly-used MCUs. If this approach is proven effective, it may realize a "write once, run everywhere" at language level, with interoperability provided one layer below the middleware. At the same time, there is a push from commercial players in the networking arena to bring IP-based solutions in WSNs (e.g., through the 6LOWPAN and ROLL initiatives [16]). These efforts aim at simplifying interoperability across and outside WSNs by relying on variations of Internet technology. It is unclear, however, whether this approach is beneficial when applied inside the WSN: some key aspects of WSN networking are fundamentally different (e.g., the predominance of many-to-one communication, instead of the Internet's one-to-one or one-to-many).

Nevertheless, an IP-based networking layer could simplify the aforementioned goal of integrating the WSN into mainstream information system, through standardized protocols and services. Moreover, unlike today's WSNs, IoT

scenarios demand interoperability, as they envision nodes from different vendors, managed by different users, to interoperate seamlessly.

6 A Final Note: Meeting (and Enduring) the Real World

Designing WSN middleware is an open research topic with significant challenges, from conceptual ones such as defining abstractions to simplify programming, down to the system optimizations necessary to cope with resource limitations. In conclusion, however, we want to draw the reader's attention on two intertwined points we deem crucial.

To really affect the state of the art, *WSN middleware must concretely demonstrate its real-world applicability*. Research on WSNs is eminently system-oriented, slowly moving from labs into full-blown applications. A number of networking protocols exist that proved to sustain the challenges of real deployments. Very few WSN middleware systems can claim the same. Reverting the trend entails not only an in-field validation, but also a different mindset when designing and implementing WSN middleware: one that takes into consideration since the beginning the trade-offs between research speculation and practical issues.

Therefore, *the design of WSN middleware cannot be addressed as "business as usual"*. Due to resource constraints and the specialized nature of WSN applications, middleware designs must take into account aspects typically disregarded in mainstream middleware. For instance, a middleware for large-scale distributed computing can be successfully designed without knowledge of the underlying MAC protocols: the same does not hold for WSNs. Designing and implementing WSN middleware requires a broad blend of competences that vertically span several layers of the stack and, equally important, intersect numerous research communities, ultimately calling for a concerted effort.

References

1. makeSense—Easy Programming of Integrated Wireless Sensor Networks. www.project-makesense.eu
2. TinyOS TEP 126 - CC2420 radio stack. www.tinyos.net/tinyos-2.x/doc/html/tep126.html
3. TeenyLIME Web site. teenylime.sf.net
4. Akyildiz, I., Su, W., Sankarasubramaniam, Y., Cayirci, E.: A survey on sensor networks. IEEE Communication Mag. **40**(8) (2002)
5. Brouwers, N., Langendoen, K., Corke, P.: Darjeeling, a feature-rich VM for the resource poor. In: Proc. of the 7th Conf. on Embedded Networked Sensor Systems (SENSYS) (2009)
6. Business Process Management Initiative: www.bpmn.org
7. Ceriotti, M., Corrà, M., D'Orazio, L., Doriguzzi, R., Facchin, D., Guna, S., Jesi, G., Lo Cigno, R., Mottola, L., Murphy, A., Pescalli, M., Picco, G.P., Pregnolato, D., Torghele, C.: Is There Light at the Ends of the Tunnel? Wireless Sensor Networks for Adaptive Lighting in Road Tunnels. In: Proc. of the 10th Int. Conf. on Information Processing in Sensor Networks (IPSN) (2011)
8. Ceriotti, M., Mottola, L., Picco, G.P., Murphy, A.L., Guna, S., Corrà, M., Pozzi, M., Zonta, D., Zanon, P.: Monitoring heritage buildings with wireless sensor networks: The Torre Aquila deployment. In: Proc. of the 8th Int. Conf. on Information Processing in Sensor Networks (IPSN) (2009)
9. Chatzigiannakis, I., Mylonas, G., Nikoletseas, S.: 50 ways to build your application: A survey of middleware and systems for wireless sensor networks. In: Proc. of the Int. Conf. on Emerging Technologies and Factory Automation (ETFA) (2007)
10. Choi, J.I., Kazandjieva, M., Jain, M., Levis, P.: The case for a network protocol isolation layer. In: Proc. of the 7th Conf. on Embedded Networked Sensor Systems (SENSYS) (2009)
11. Costa, P., Coulson, G., Gold, R., Lad, M., Mascolo, C., Mottola, L., Picco, G.P., Sivaharan, T., Weerasinghe, N., Zachariadis, S.: The RUNES middleware for networked embedded systems and its application in a disaster management scenario. In: Proc. of the 5th Int. Conf. on Pervasive Communications (PerCom) (2007)
12. Costa, P., Mottola, L., Murphy, A.L., Picco, G.P.: Programming wireless sensor networks with the TeenyLIME middleware. In: Proc. of the 8th ACM/USENIX Int. Middleware Conf. (2007)
13. Henricksen, K., Robinson, R.: A survey of middleware for sensor networks: state-of-the-art and future directions. In: Proc. of the 1st ACM Int. Workshop on Middleware for Sensor Networks (MidSens) (2006)
14. Hill, J., Culler, D.: Mica: A wireless platform for deeply embedded networks. IEEE Micro **22** (2002)
15. Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D., Pister, K.: System architecture directions for networked sensors. In: Proc. of the 9th Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX) (2000)
16. IP for Smart Objects Alliance: ipso-alliance.org
17. Juang, P., Oki, H., Wang, Y., Martonosi, M., Peh, L.S., Rubenstein, D.: Energy-efficient computing for wildlife tracking: Design tradeoffs and early experiences with ZebraNet. SIGPLAN Not. **37**(10) (2002)
18. Levis, P., Madden, S., Gay, D., Polastre, J., Szewczyk, R., Woo, A., Brewer, E., Culler, D.: The emergence of networking abstractions and techniques in TinyOS. In: Proc. of 1st Symp. on Networked System Design and Implementation (NSDI) (2004)
19. Liu, T., Martonosi, M.: Impala: A middleware system for managing autonomic, parallel sensor systems. In: Proc. of the 9th Symp. on Principles and Practice of Parallel Programming (2003)
20. Madden, S., M.J. Franklin, J.M. Hellerstein, Hong, W.: TinyDB: An acquisitional query processing system for sensor networks. ACM Trans. on Database Systems **30**(1) (2005)
21. Marrón, P.J., Gauger, M., Lachenmann, A., Minder, D., Saukh, O., Rothermel, K.: FlexCup: A flexible and efficient code update mechanism for sensor networks. In: Proc. of the 3rd European Workshop on Wireless Sensor Networks (EWSN) (2006)
22. Mascolo, C., Capra, L., Emmerich, W.: Mobile computing middleware. Advanced lectures on networking (2002)
23. Mottola, L., Picco, G.P.: Programming Wireless Sensor Networks: Fundamental Concepts and State of the Art. ACM Computing Surveys **43**(11) (2011)
24. Mottola, L., Picco, G.P., Amjad, A.: Fine-grained software reconfiguration in wireless sensor networks. In: Proc. of 5th European Conf. on Wireless Sensor Networks (EWSN) (2008)
25. Polastre, J., Szewczyk, R., Culler, D.: Telos: enabling ultra-low power wireless research. In: Proc. of the 5th Int. Conf. on Information Processing in Sensor Networks (IPSN) (2005)
26. Römer, K.: Programming paradigms and middleware for sensor networks. In: GI/ITG Workshop on Sensor Networks (2004)
27. Sugihara, R., Gupta, R.K.: Programming models for sensor networks: A survey. ACM Trans. on Sensor Networks **4**(2) (2008)
28. Wang, M.M., Cao, J., Li, J., Das, S.: Middleware for wireless sensor networks: A survey. J. of Computer Science and Technology **23**(3), 305–326 (2008)
29. ZigBee Alliance: www.zigbee.org